Laboratorium grafowe 3

Projekt „Matematyka dla Ciekawych Świata”,
Tomasz Świerczewski, Jakub Jałowiec, Łukasz Mazurek

19.04.2021

1 Wstęp

W tym skrypcie będziemy się uczyć jak zaimplementować grafy w Pythonie. W tym celu nie pojawi się początkowo w tym skrypcie gotowe rozwiązanie jak to zrobić. Po przerobieniu danego zagadnienia po zakończeniu zajęć prowadzący prześle wersję z tym co było zrobione na ćwiczeniach.

2 Dodawanie wierzchołków oraz krawędzi do grafu

W poprzednim skrypcie omówiliśmy naszą podstawową implementację grafów wykorzystującą słowniki oraz listy. W następnym kroku postaramy się dodać kilka funkcji, które pozwolą nam na modyfikacje stworzonego grafu, na przykład poprzez dodawanie nowych wierzchołków oraz krawędzi do grafu.

Przećwiczmy kolejno:

1. dodawanie nowych wierzchołków do grafu
2. dodawanie pojedynczej krawędzi
3. dodawanie listy krawędzi
4. dodawanie wierzchołka i krawędzi naraz
5. zliczanie krawędzi

Ćwiczenie 2.0.1

Napisz funkcję `dodaj_wierzcholek(graf, wierzcholek)`, która przyjmuje 2 argumenty: `graf` do modyfikacji oraz nowo dodawany `wierzchołek` do grafu. Dodaj tylko sam wierzchołek z pustą listą sąsiedztwa.

Ćwiczenie 2.0.2

Napisz funkcję `dodaj_krawedz(graf, zrodlo, cel)`, która przyjmuje 3 argumenty: graf do modyfikacji oraz oraz dwa wierzchołki. Funkcja ma za zadanie dodać nową krawędź do tego grafu. Pamiętaj o tym, że jest to graf nieskierowany, a więc musimy dodać nowy element do listy dla dwóch kluczy w słowniku.

Ćwiczenie 2.0.3

Napisz funkcję `dodaj_wierzcholek_z_krawedziami(graf, wierzcholek, krawedzie)`, która przyjmuje 3 argumenty: graf do modyfikacji, nowo dodawany wierzchołek do grafu oraz listę z wierzchołkami, do których mają być poprowadzone krawędzie od nowo dodawanego wierzchołka. Pamiętaj o tym, że jest to graf nieskierowany, zatem musimy poprowadzić krawędzie zarówno od jak i do nowo dodawanego wierzchołka.

Ćwiczenie 2.0.4

Napisz funkcję `dodaj_wierzcholek_z_krawedziami_do_wszystkich(graf, wierzcholek)`, która doda podany `wierzchołek` do `grafu` oraz połączy go z wszystkimi wierzchołkami, które były dotychczas w grafie.

Ćwiczenie 2.0.5

Napisz funkcję `zlicz_krawedzie(graf)`, która zliczy łączną liczbę krawędzi w grafie.

Dla dociekliwych

Graf „nieskierowany” a graf „skierowany”

Często rozróżnia się grafy *skierowane* i grafy *nierskierowane*. Grafy *skierowane* to takie, w których kierunek krawędzi ma znaczenie, tzn. jeśli istnieje krawędź z wierzchołka `X` do `Y`, to niekoniecznie istnieje krawędź z wierzchołka `Y` do wierzchołka `X`.

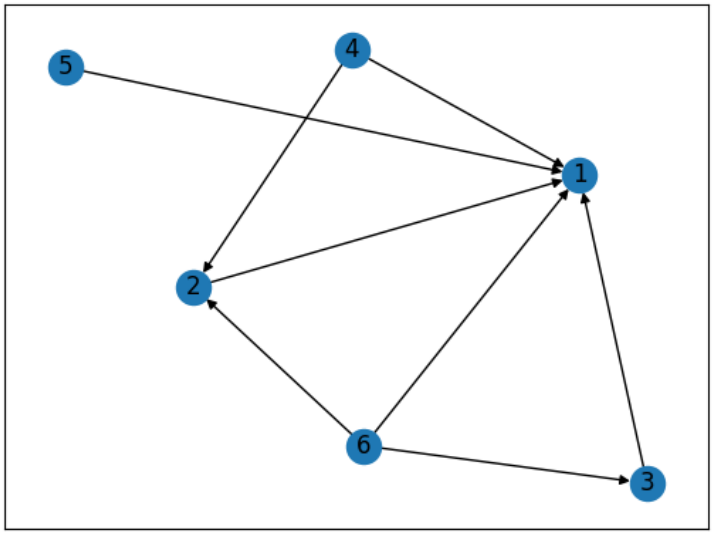
Natomiast grafy *nieskierowane* to takie grafy, w których kierunek krawędzi nie ma znaczenia. Często przyjmuje się dla uproszczenia, że w grafach *nieskierowanych* jeśli istnieje krawędź z wierzchołka `X` do `Y`, to istnieje również krawędź z wierzchołka `Y` do wierzchołka `X`.

Zadanie 2.0.1

Napisz funkcję, która przyjmuje jeden argument — liczbę, nazwijmy ją `n` i stworzy graf, który ma `n` wierzchołków i obrazuje podzielność liczb. Jeśli np. liczba 6 jest podzielna przez 1, 2 i 3, to będzie miała krawędzie skierowane do tych liczb, przez które jest podzielna. Pomijamy podzielność liczby przez samą siebie.

Np. dla zakresu do 6 będziemy mieli wierzchołki 1, 2, 3, 4, 5 i 6, wierzchołki 2, 3, 4, 5, 6 są połączone do 1, wierzchołki 4 i 6 do 2, wierzchołek 6 także do 3. Najlepiej to obrazuje rysunek 1.

Rysunek 1: Przykładowy wynik tej funkcji dla argumentu równego 6:

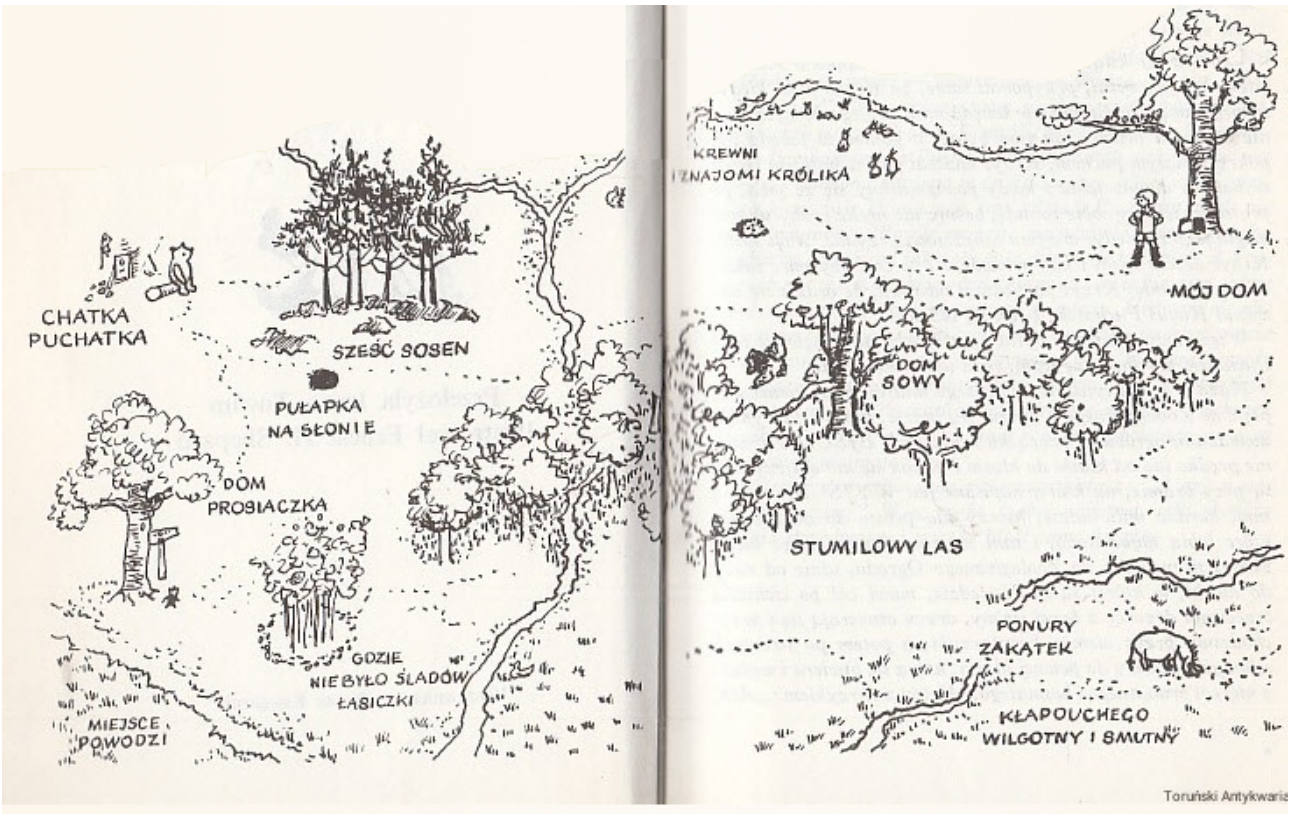


3 Przeszukiwanie grafu

3.1 Stumilowy Las

Jednymi z pierwszych algorytmów jakie poznamy będzie *przeszukiwanie grafu*. Na początek zajmiemy się *wyszukiwaniem wszerz* - znanym po angielsku jako *Breadth-first search* (*BFS*). Do naszych rozważań na temat przeszukiwania grafu wykorzystamy fragment Stumilowego Lasu widoczny na Rysunku 2.

Rysunek 2: Fragment Stumilowego Lasu. Niech wierzchołkami grafu będą miejsca w Stumilowym Lesie, a krawędziami grafu ścieżki między miejscami.



Miejsca i ścieżki w Stumilowym Lesie zostały przedstawione jako następujące wierzchołki:

Tabela 1: Miejsca i ścieżki w Stumilowym Lesie

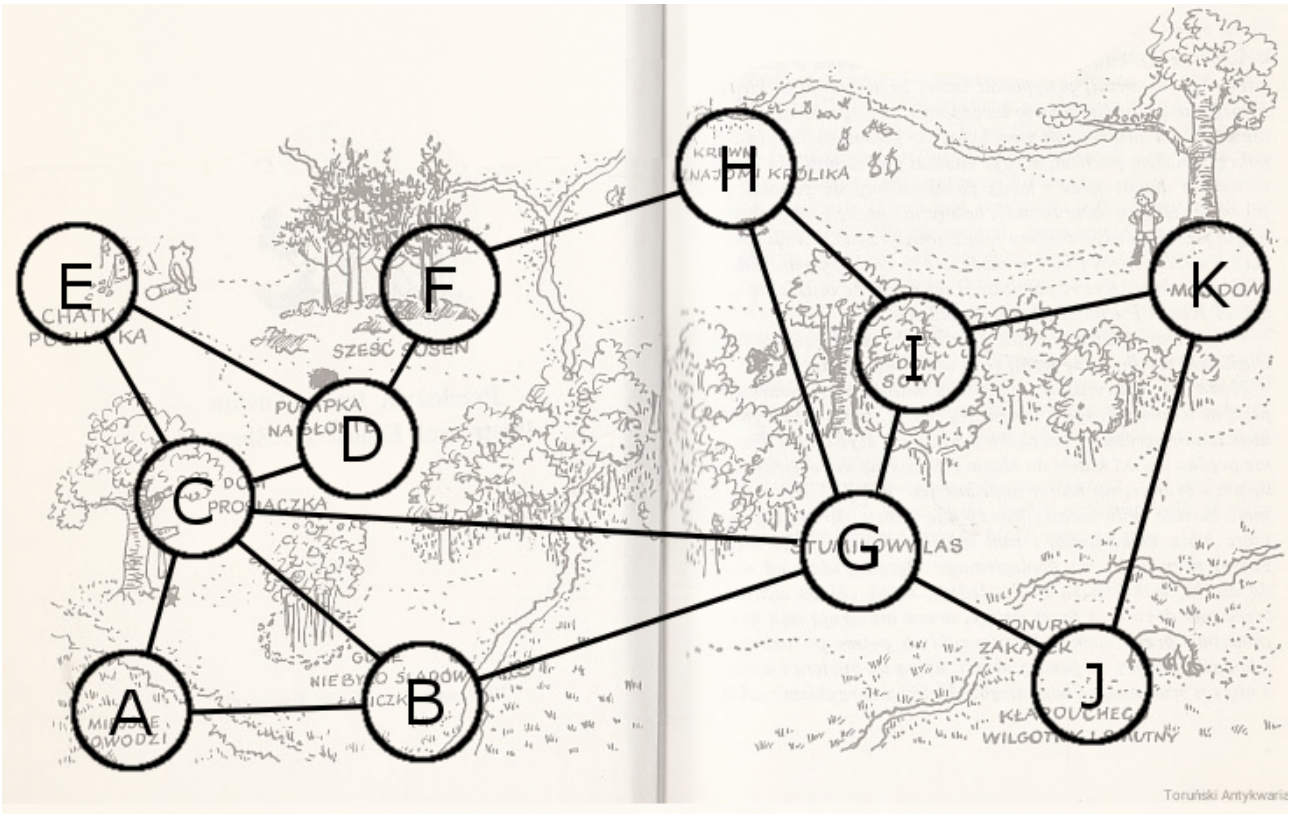
Miejsce	**Ma ścieżkę do...**
Miejsce powodzi	Gdzie nie było śladów łasiczki, Dom Prosiaczka
Gdzie nie było śladów łasiczki	Miejsce powodzi, Dom Prosiaczka, Stumilowy Las
Dom Prosiaczka	Miejsce powodzi, Gdzie nie było śladów łasiczki, Pułapka na słonie, Chatka Puchatka
Pułapka na słonie	Dom Prosiaczka, Chatka Puchatka, Sześć sosen
Chatka Puchatka	Dom Prosiaczka, Pułapka na słonie
Sześć sosen	Pułapka na słonie, Krewni i znajomi Królika
Stumilowy Las	Gdzie nie było śladów łasiczki, Dom Prosiaczka, Krewni i znajomi Królika, Dom Sowy, Zakątek Kłapouchego
Dom Sowy	Krewni i znajomi Królika, Stumilowy Las, Dom Krzysia
Zakątek Kłapouchego	Stumilowy Las, Dom Krzysia

Skoro już opisaliśmy dosyć dokładnie fragment Stumilowego Lasu to możemy teraz przedstawić go w formie grafu nieskierowanego. Na Rysunku 3 przedstawiony jest jeszcze raz ten sam fragment Stumilowego

Lasu, ale tym razem z wyróżnionymi miejscami jako wierzchołkami i ścieżkami jako krawędziami między wierzchołkami.

Na Rysunku 3 widoczny jest graf utworzony z mieszkań i ścieżek między miejscami w Stumilowym Lesie.

Rysunek 3: Graf nieskierowany powstały z miejsc i ścieżek w Stumilowym Lesie.



Graf z Rysunku 3 możemy zapisać w języku Python jako następujący słownik :

```
stumilowy_las = {
    "A": ["B", "C"],
    "B": ["A", "C", "G"],
    "C": ["A", "B", "D", "E", "G"],
    "D": ["C", "E", "F"],
    "E" : ["C", "D"],
    "F" : ["D", "H"],
    "G": ["B", "C", "H", "I", "J"],
    "H": ["F", "G", "I"],
    "I": ["G", "H", "J", "K"],
    "J" : ["G", "K"],
    "K" : ["I", "J"]
}
```

3.2 Przeszukiwanie wszerz

Działanie przeszukiwania wszerz polega na tym, że odwiedzamy w grafie najpierw wszystkich bezpośrednich sąsiadów *wierzchołka początkowego*, potem wszystkich sąsiadów jego sąsiadów, potem wszystkich sąsiadów sąsiadów jego sąsiadów itd. Kolejne poziomy sąsiedztwa wyznaczają "warstwy poziome" grafu. "Warstwy" te wyznaczone są przez odległość danego wierzchołka od *wierzchołka początkowego*, stąd nazwa - *przeszukiwanie wszerz*.

Poniżej przedstawiony jest pseudokod *przeszukiwania wszerz*, a przeszukiwanie grafu fragmentu Stumilowego Lasu jest rozrysowane na Rysunku 4 i w Tabeli 2.

```
0. Funkcja przeszukiwanie_wszerz(graf, poczatek):
1.   Dla każdego wierzchołka w grafie:
2.     Oznacz, że nie odwiedziliśmy tego wierzchołka.
3.   Dodaj do kolejki wierzchołek początkowy i oznacz, że został odwiedzony.
4.   Dopóki kolejka nie jest pusta:
5.     Pobierz pierwszy wierzchołek z kolejki.
6.     Dla każdego sąsiada tego wierzchołka:
7.       Jeśli cel nie był jeszcze odwiedzony:
8.         Oznacz go jako odwiedzony.
9.         Dodaj go na koniec kolejki.
```

3.3 Przeszukiwanie w głąb

Jaki byłby efekt, gdybyśmy w powyższym listingu zmienili linijkę nr 9. z "`Dodaj go na koniec kolejki`" na "`Dodaj go na początek kolejki`". Całkowicie przez to zmieni się kolejność odwiedzania wierzchołków - zamiast przechodzenia grafu po kolejnych "warstwach" zaczynając od najbliższej wierzchołki będą odwiedzane "na oślep, byle do przodu". Taki sposób przeglądania nazywamy *w głąb*.

Na Rysunku 5 oraz w Tabeli 3 rozpisane jest *przeszukiwanie w głąb* dla Stumilowego Lasu. Tak naprawdę w *przeszukiwaniu w głąb* nie są już istotne "warstwy" znane nam z *przeszukiwania wszerz*, ale na Rysunku 5 nadal są one zaznaczone, aby móc porównać oba algorytmy.

Dla dociekliwych

Przeszukiwanie *w głąb* a przeszukiwanie *wszerz*
Zmiana jednej linijki w algorytmie daje zupełnie różne działanie algorytmów. W *przeszukiwaniu w głąb* wstawialiśmy odwiedzane wierzchołki na początek kolejki. Takie podejście znane jest jako *kolejkowanie FIFO* (ang. *First In, First Out — pierwszy na wejściu, pierwszy na wyjściu*). Przypomina w swoim zachowaniu kolejkę w sklepie. Ta osoba, która stanęła pierwsza w kolejce, będzie pierwsza obsłużona, ta co jaka ostatnia, będzie jako ostatnia. Za każdym razem dodawaliśmy na koniec elementy do kolejki, a pobieraliśmy te co były z początku.
W przeszukiwania wszerz wstawialiśmy odwiedzane wierzchołki na koniec kolejki. To podejście zwane jest kolejkowaniem LIFO (ang. *Last In, First Out — ostatni na wejściu, pierwszy na wyjściu*). W życiu codziennym można ją zilustrować naszym typowym zachowaniem w kuchni. Jeśli przez 3 dni będziemy kupować po 3 bułki i będziemy codziennie zjadać 2, to 4 dnia zostaną nam 3 bułki — po jednej z każdego dnia. Większość osób sięgnie po tę najświeższą — ostatnia kupiona zostanie pierwsza zjedzona, ponieważ jest najświeższa.

3.4 Implementacja algorytmów przeszukiwania

Na kolejnych stronach przećwiczymy zarówno *przeszukiwanie wszerz* jaka i *przeszukiwanie w głąb*.

Przeszukiwanie wszerz

Rysunek 4: Graf Stumilowego Lasu z dodanymi numerami odwiedzenia każdego z wierzchołków przy *przeglądaniu wszerz*. Przykładowo wierzchołek A został odwiedzony jako pierwszy, wierzchołek B jako drugi, wierzchołek C jako trzeci itd. Kolorami zaznaczone są kolejne warstwy.

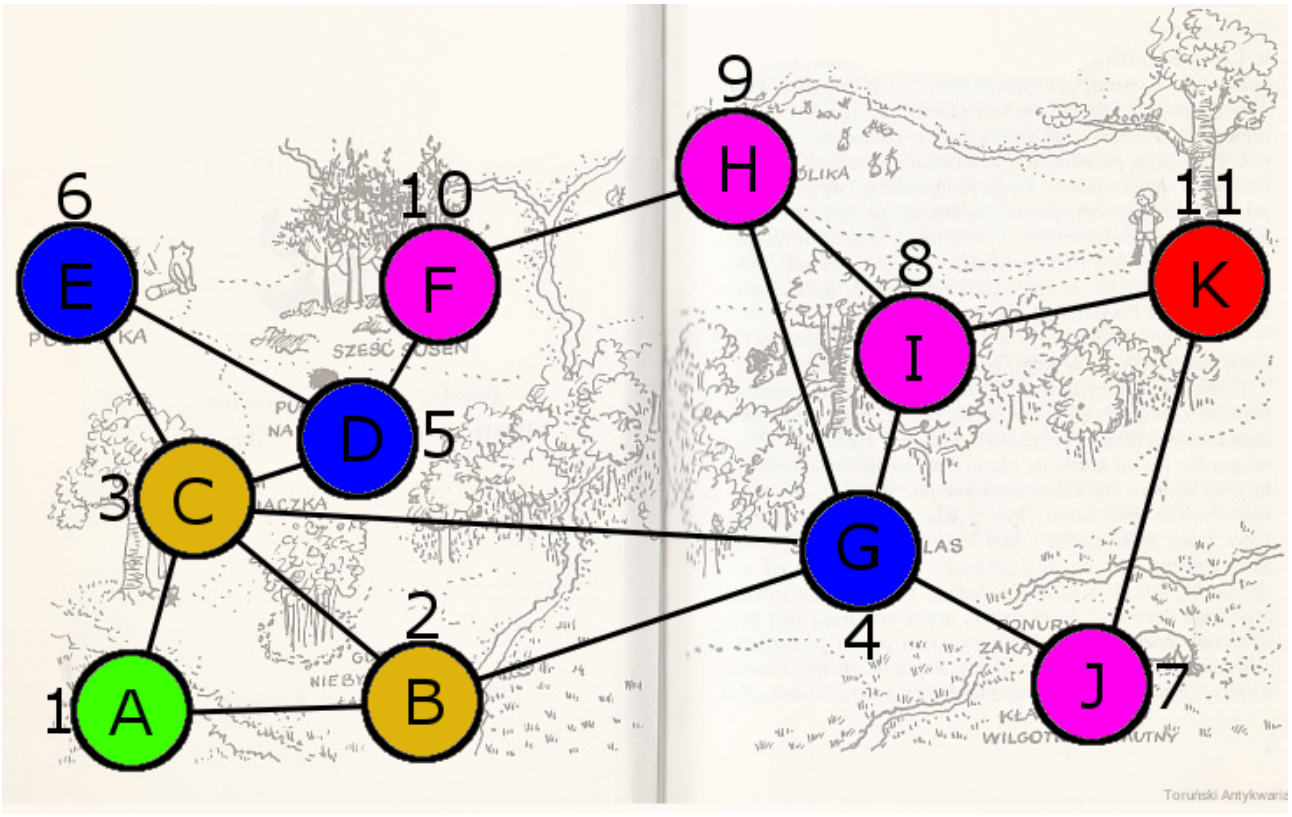


Tabela 2: Przeglądanie grafu wszerz - szczegóły każdego kroku.

iteracja	**wierzchołek roboczy**	**warstwa (odległość od `'A'`)**	**kolejka na koniec kroku**
1	`'A'`	warstwa 0	`['B', 'C']`
2	`'B'`	warstwa 1	`['C', 'G']`
3	`'C'`	warstwa 1	`['G', 'D', 'E']`
4	`'G'`	warstwa 2	`['D', 'E', 'H', 'I', 'J']`
5	`'D'`	warstwa 2	`['E', 'H', 'I', 'J', 'F']`
6	`'E'`	warstwa 2	`['H', 'I', 'J', 'F']`
7	`'H'`	warstwa 3	`['I', 'J', 'F']`
8	`'I'`	warstwa 3	`['J', 'F', 'K']`
9	`'J'`	warstwa 3	`['F', 'K']`
10	`'F'`	warstwa 3	`['K']`
11	`'K'`	warstwa 4	`[]`

Ćwiczenie 3.4.1

Napisz funkcję `przeszukiwanie_wszerz(graf, poczatek)`, implementującą algorytm *przeszukiwania wszerz* grafu Lasu Stumilowego. Przykładowe wywołanie:

```
> # stumilowy_las = {...}
> przeszukiwanie_wszerz(stumilowy_las, 'A')
Kolejnosc odwiedzenia: A, B, C, G, D, E, H, I, J, F, K
```

Przeszukiwanie w głąb

Rysunek 5: Graf Stumilowego Lasu z dodanymi numerami odwiedzenia wierzchołków przeglądając *w głąb*. W *przeszukiwaniu w głąb* nie są istotne warstwy grafu, ale zostały zaznaczone dla porównania z *przeszukiwaniem wszerz*.

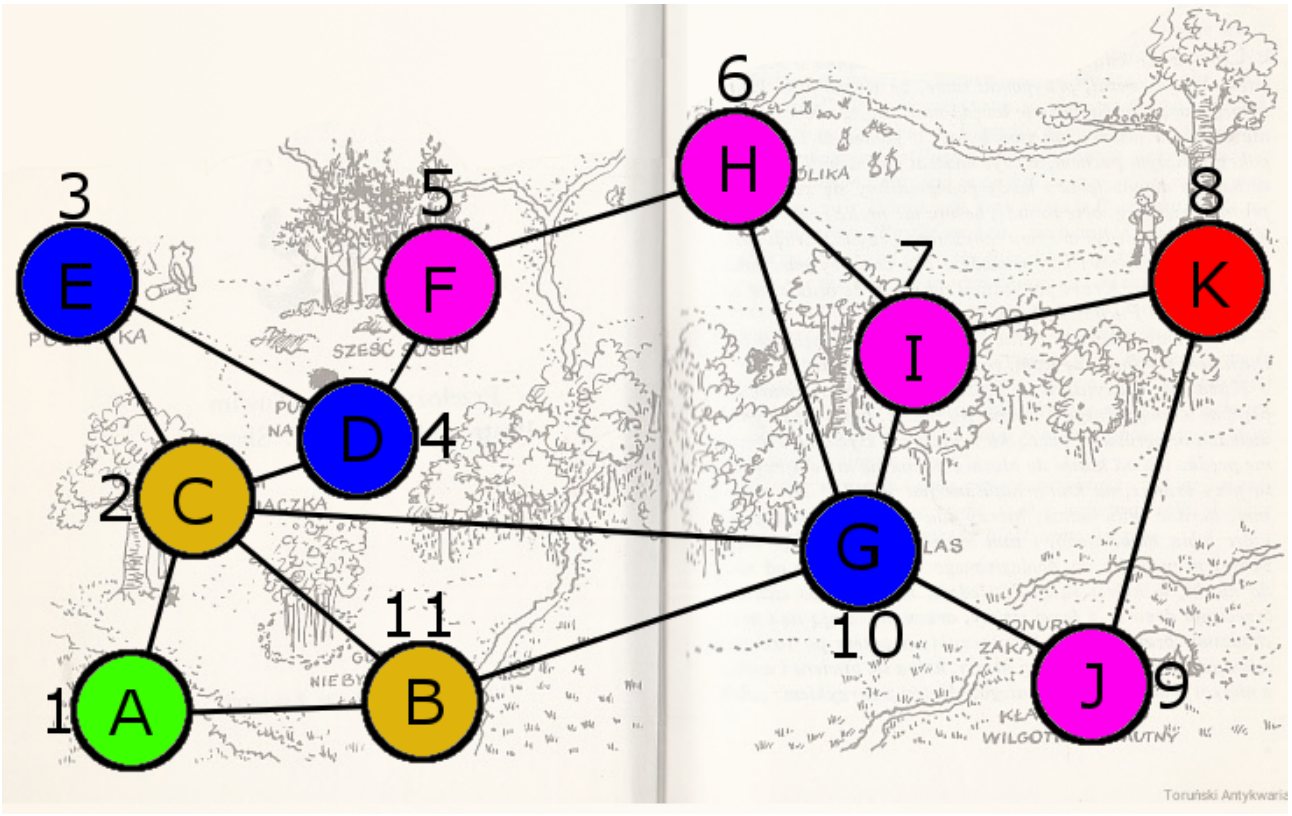


Tabela 3: Przeglądanie grafu w głąb - szczegóły każdego kroku.

iteracja	wierzchołek roboczy	kolejka na koniec kroku
1	`'A'`	`['C', 'B']`
2	`'C'`	`['E', 'D', 'B']`
3	`'E'`	`['D', 'B']`
4	`'D'`	`['F', 'B']`
5	`'F'`	`['H', 'B']`
6	`'H'`	`['I', 'G', 'B']`
7	`'I'`	`['K', 'G', 'B']`
8	`'K'`	`['J', 'G', 'B']`
9	`'J'`	`['G', 'B']`
10	`'G'`	`['B']`
11	`'B'`	`[]`

Ćwiczenie 3.4.2

Napisz funkcję `przeszukiwanie_w_glab(graf, poczatek)`, implementującą algorytm *przeszukiwania w głąb* grafu Lasu Stumilowego. Przykładowe wywołanie:

```
> # stumilowy_las = {...}
> przeszukiwanie_w_glab(stumilowy_las, 'A')
Kolejnosc odwiedzenia: A, C, E, D, F, H, I, K, J, G, B
```

4 Zadania

Zadanie 4.0.1

Prosiaczek chce odwiedzić Krzysia i potem udać się razem z nim na poszukiwania łasiczki do *Miejsca, gdzie jej nie było*. Napisz funkcję, którą dobiera możliwie krótkie ścieżki, aby to zrobić. Funkcja ma wypisywać na ekran ścieżkę do Krzysia, a potem ścieżkę do miejsca poszukiwania łasiczki. Przykładowe wywołanie:

```
> stumilowy_las = {...}
> poszukiwanie_lasiczki(stumilowy_las, 'C', 'K', 'A')
Sciezka Prosiaczek -> Krzys: C, G, I, K
Sciezka Krzys i Prosiaczek -> lasiczka: K, I, G, B
```

Zadanie 4.0.2

W Stumilowym Lesie jest noc, a dwa Hefalumpy wpadły w *Pułapkę na słonie*. Postanowiły z niej uciec do *Miejsca, gdzie nie było powodzi*, ale aby zmylić trop, gdy obudzą się mieszkańców Stumilowego Lasu umówiły się, że się rozdzielą - jeden Hefalump pobiegnie pierwszy, a gdy będzie bezpieczny to zatrąbi, dając znać drugiemu, że on też może już uciec. Drugi Hefalump nie może jednak pobiec żadną ścieżką, którą biegł już pierwszy Hefalump. Jak powinny pobiec oba Hefalumpy, aby musiały przebiec najkrótszą możliwą drogę?

Napisz funkcję `ucieczka_hefalumpow(graf, zrodlo, cel)`, która przyjmuje jako argument dowolny `graf` reprezentowany jako słownik, `zrodlo` będące jednym z wierzchołków `grafu` (domyślnie niech to będzie *Pułapka na słonie*), a `cel` to wierzchołek, w którym oba Hefalumpy na koniec chcą się schronić (domyślnie to *Miejsce, gdzie nie było powodzi*). Funkcja ma wypisywać ścieżkę pierwszego Hefalumpa oraz ścieżkę drugiego Hefalumpa.

Przykładowe wywołanie:

```
> # stumilowy_las = {...}
> ucieczka_hefalumpow(stumilowy_las, 'D', 'A')
Pierwszy Hefalump: C, A
Drugi Hefalump: D, F, H, G, B, A
```

Zadanie 4.0.3

Kubuś Puchatek planuje spacer po Stumilowym Lesie. Stwierdził, że dla urozmaicenia chce chodzić w taki sposób po Lesie, że kolejne miejsca odwiedzane przez niego zawierają kolejne litery wybranego wzorca. Jedno miejsce może odwiedzić wielokrotnie. Przykładowo, jeśli wybrał wzorzec `"miodek"` to może przejść ścieżką `['Dom Prosiaczka', 'Stumilowy Las', 'Zakatek Klapouchego', 'Dom Krzysia', 'Zakatek Klapouchego', 'Dom Krzysia']`, bo:

- `"m"` jest w `"Dom Prosiaczka"`
- `"i"` jest w `"Stumilowy Las"`
- `"o"` jest w `"Zakatek Klapouchego"`
- `"d"` jest w `"Dom Krzysia"`
- `"e"` jest w `"Zakatek Klapouchego"`
- `"k"` jest w `"Dom Krzysia"`

Napisz funkcję `znajdz_wzorzec(graf, wzorzec)`, która znajduje ścieżkę, przechodzącą po kolei przez miejsca, które zawierają kolejne litery z *wzorca*. Przykładowe wywołanie:

```
> stumilowy_las = {...}
> znajdz_wzorzec(stumilowy_las, "miodek")
Sciezka: ['Dom Prosiaczka', 'Stumilowy Las', 'Zakatek Klapouchego', 'Dom
 ↪  Krzysia', 'Zakatek Klapouchego', 'Dom Krzysia']
> znajdz_wzorzec(stumilowy_las, "abcde")
Sciezka: ['Dom Prosiaczka', 'Gdzie nie bylo sladow lasiczki', 'Dom Prosiaczka',
 ↪  'Gdzie nie bylo sladow lasiczki', 'Miejsce powodzi']
> znajdz_wzorzec(stumilowy_las, "abcdef")
Nie znaleziono sciezki
```

Praca domowa nr 3

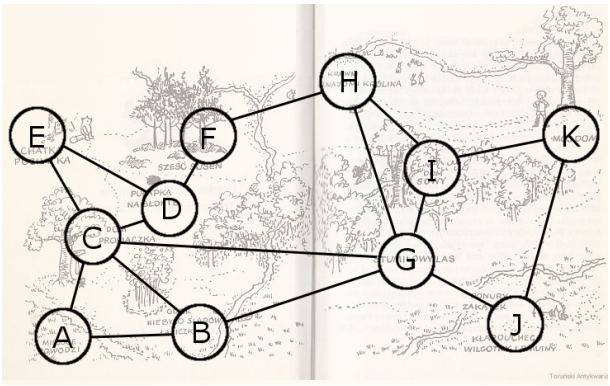
Uwaga. O ile nie przekazano na zajęciach inaczej to pracę domową należy przesłać prowadzącym w wiadomości prywatnej na Discordzie.

Zadanie 4.0.4 — 2 pkt

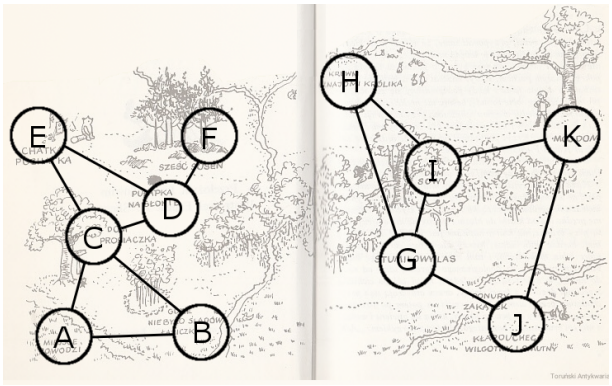
Napisz funkcję `czy_spojny(graf)`, która sprawdzi czy podany graf nieskierowany (taki, w którym jeśli istnieje krawędź `A->B`, to istnieje krawędź `B->A`) jest spójny. Funkcja ma przyjmować jako argument graf reprezentowany jako słownik list, a zwracać wartość logiczną `True` jeśli podany graf jest spójny lub `False` w przeciwnym przypadku. Graf jest spójny wtedy i tylko wtedy, gdy uruchamiając algorytm przeszukiwania w dowolnym wierzchołku możemy odwiedzić wszystkie inne wierzchołki.

Przykład:

```
> stumilowy = {"A": ["B", "C"], "B": ["A", "C", "G"], "C": ["A", "B", "D", "E",
 ↪  "G"], "D": ["C", "E", "F"], "E" : ["C", "D"], "F" : ["D", "H"], "G": ["B",
 ↪  "C", "H", "I", "J"], "H": ["F", "G", "I"], "I": ["G", "H", "J", "K"], "J" :
 ↪  ["G", "K"], "K" : ["I", "J"]
}
> czy_spojny(stumilowy_1)
True
>
> stumilowy_2 = {"A": ["B", "C"], "B": ["A", "C"], "C": ["A", "B", "D", "E"],
 ↪  "D": ["C", "E", "F"], "E" : ["C", "D"], "F" : ["D"], "G": ["H", "I", "J"],
 ↪  "H": ["G", "I"], "I": ["G", "H", "J", "K"], "J" : ["G", "K"], "K" : ["I",
 ↪  "J"]
}
> czy_spojny(stumilowy_2)
False
```



(a) Graf spójny (`stumilowy_1` z przykładu wyżej).



(b) Graf niespójny (`stumilowy_2` z przykładu wyżej).

Zadanie 4.0.5 — 1 pkt

Napisz funkcję `stopien_wierzcholka(graf, wierzcholek)`, która przyjmuje dwa argumenty: `graf` reprezentowany jako słownik list oraz `wierzcholek`, bedący pojedynczą wartością i zwracającą *stopień* tego `wierzchołka`. Za *stopień wierzchołka* uznajemy liczbę krawędzi wychodzących od danego wierzchołka.

Przykład:

```
> # stumilowy_las = {...}
> stopien_wierzcholka(stumilowy_las, "A")
2
> stopien_wierzcholka(stumilowy_las, "B")
3
> stopien_wierzcholka(stumilowy_las, "G")
5
> stopien_wierzcholka(stumilowy_las, "K")
2
```

Zadanie 4.0.6 — 1 pkt

Napisz funkcję `stopien_wchodzacy_wierzcholka(graf, wierzcholek`, przyjmującą jako argumenty `graf` reprezentowany jako słownik list oraz `wierzcholek`, będący pojedynczą wartością i zwracającą stopień wchodzący wierzchołka w grafie skierowanym, czyli liczbę krawędzi wchodzących do danego wierzchołka.

Przykład:

```
> graf = { 1: [2, 3], 2: [1, 3, 4], 3: [1, 2], 4: [1] }
> stopien_wchodzacy_wierzcholka(graf, 1)
3
> stopien_wchodzacy_wierzcholka(graf, 2)
2
> stopien_wchodzacy_wierzcholka(graf, 3)
2
> stopien_wchodzacy_wierzcholka(graf, 4)
1
```

Zadanie 4.0.7 — 1 pkt

Napisz funkcję `usun_krawedz(graf, poczatek, koniec)`, która przyjmuje jako argumenty `graf` reprezentowany jako słownik list, `poczatek` będący pojedynczą wartością i `koniec`, będący pojedynczą wartością i usuwającą krawędź między wierzchołkiem `poczatek`, a wierzchołkiem `koniec` (jeśli taka istnieje).

Przykład:

```
> graf = { 1: [2, 3], 2: [1, 3, 4], 3: [1, 2], 4: [1] }
> usun_krawedz(graf, 1, 3)
> graf
{1: [2], 2: [1, 3, 4], 3: [1, 2], 4: [1]}
```

Podpowiedź: Z listy możemy usuwać elementy w następujący sposób:

```
> lista = [1, 2, 3]
> lista.remove(2)
> lista
[1, 3]
```

Zadanie 4.0.8 — 1 pkt

Napisz funkcję `usun_wierzcholek(graf, wierzcholek)`, przyjmującą jako argumenty `graf`, reprezentowany jako słownik oraz `wierzcholek`, będący pojedynczą wartością i usuwającą zadany wierzchołek z grafu, a także wszystkie krawędzie, które łączyły się z tym wierzchołkiem.

Przykład:

```
> graf = { 1: [2, 3], 2: [1, 3, 4], 3: [1, 2], 4: [1] }
> usun_wierzcholek(graf, 4)
> graf
{1: [2, 3], 2: [1, 3], 3: [1, 2]}
```

Podpowiedź: Przydatne jest wykonanie najpierw zadania 5.0.4. Ponadto, ze słownika możemy usuwać klucz w następujący sposób używając słowa kluczowego **`del`** *(z ang. delete, usuń):*

```
> slownik = {1: [2, 3], 2: [1, 3], 3: [1, 2]}
> del slownik[1]
> slownik
{2: [1, 3], 3: [1, 2]}
```

Zadanie 4.0.9 — 1 pkt

Napisz funkcję `czy_drzewo(graf)`, przyjmującą jako jedyny argument `graf` nieskierowany (taki, w którym jeśli istnieje krawędź `A->B`, to istnieje krawędź `B->A`), reprezentowany jako słownik list i sprawdzającą czy jest on *drzewem*. Drzewo to acykliczny i spójny graf.

Przykład:

```
> graf1 = {1: [2], 2: [1, 3], 3: [2, 4], 4: [3]}
> czy_drzewo(graf1)
True
> graf2 = {1: [2], 2: [1, 3, 4], 3: [2, 4], 4: [2, 3]}
> czy_drzewo(graf2)
False
```

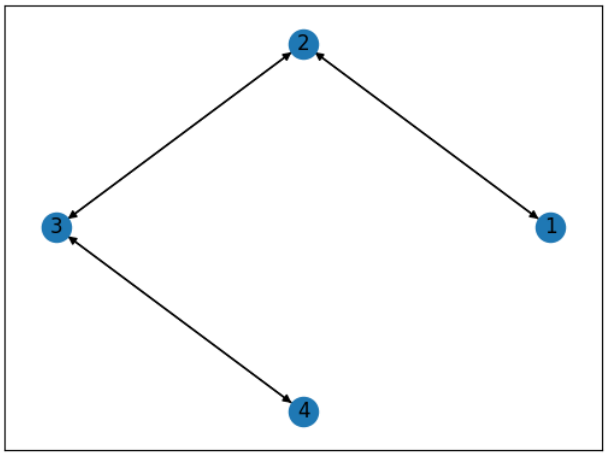
Podpowiedź: Zrób najpierw zadania 5.0.1 oraz 5.0.6.

Zadanie 4.0.10 — 2 pkt

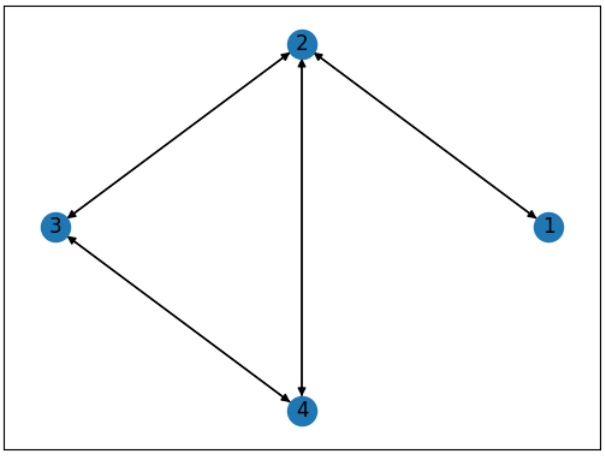
Napisz funkcję `czy_ma_cykl(graf)`, przyjmującą jako argument `graf` nieskierowany (taki, w którym jeśli istnieje krawędź `A->B`, to istnieje krawędź `B->A`) reprezentowany jako słownik list i sprawdzającą czy ma on *cykl*. *Cykl* to taka sekwencja krawędzi w grafie, że pierwszy i ostatni wierzchołek tej sekwencji są identyczne.

Przykład:

```
> graf1 = {1: [2], 2: [1, 3], 3: [2, 4], 4: [3]}
> czy_ma_cykl(graf1)
False
> graf2 = {1: [2], 2: [1, 3, 4], 3: [2, 4], 4: [2, 3]}
> czy_ma_cykl(graf2)
True
```



(c) Graf bez cyklu (`graf1` z przykładu wyżej).



(d) Graf z cyklem `2->3->4->2` (`graf2` z przykładu wyżej).

Podpowiedź: cykl można wykryć używając przeszukiwania wszerz i sprawdzając czy osiągalny jest wierzchołek już raz odwiedzony.

Zadanie 4.0.11 — 2 pkt

Napisz funkcję `czy_n_regularny(graf, n)`, przyjmującą jako argument `graf` reprezentowany jako słownik list oraz `n`, będący pojedynczą liczbą całkowitą, która sprawdzi czy graf jest *n-regularny*. Graf *n-regularny* to taki graf, że każdy wierzchołek w grafie ma stopień dokładnie *n*. Dla przypomnienia - stopień wierzchołka to liczba krawędzi wychodzących z danego wierzchołka.

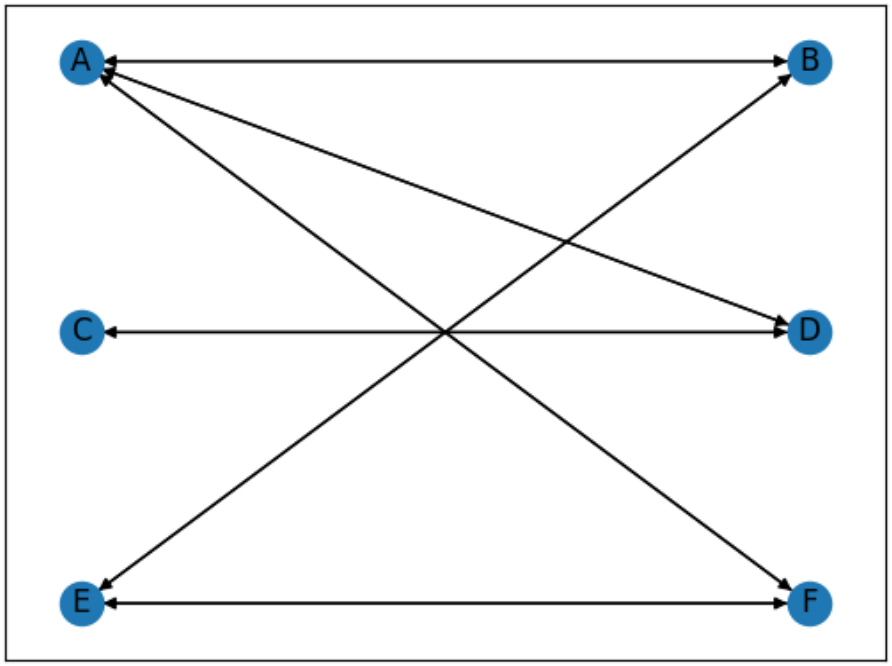
Przykład:

```
> graf = {1: [2, 3, 4], 2: [1, 3, 4], 3: [1, 2, 4], 4: [1, 2, 3]}
> czy_n_regularny(graf)
True
> graf2 = {1: [2, 3], 2: [1, 3, 4], 3: [1, 2, 4], 4: [2, 3]}
> czy_n_regularny(graf2)
False
```

Zadanie 4.0.12 — 2 pkt

Napisz funkcję `czy_dwudzielny(graf)`, przyjmującą jako argument `graf` reprezentowany jako słownik list i sprawdzającą, czy graf jest *dwudzielny*. Graf dwudzielny to taki graf, w którym jesteśmy w stanie rozdzielić jego wierzchołki na dwa rozłączne zbiory, takie że żaden wierzchołek wewnątrz danego zbioru nie łączy z innym wierzchołkiem z tego zbioru, ale może się łączyć z wierzchołkami z drugiego zbioru.

Przykładowy graf dwudzielny. Zbiór wierzchołków można rozdzielić na dwa podzbiory `{"A", "C", "E"}` oraz `{"B", "D", "F"}`, takie że między wierzchołkami należącymi do tego samego zbioru nie ma żadnej krawędzi, istnieją natomiast krawędzie między wierzchołkami z dwóch różnych zbiorów.



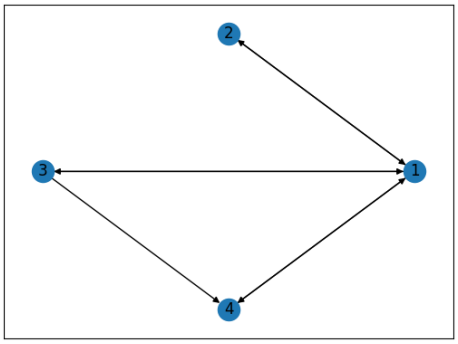
Przykład:

```
> graf1 = {"A": ["B", "D", "F"], "B": ["A", "E"], "C": ["D"], "D": ["C"], "E":
 ↪  ["B", "D", "F"], "F": ["A", "E"]}
> czy_dwudzielny(graf1)
True
```

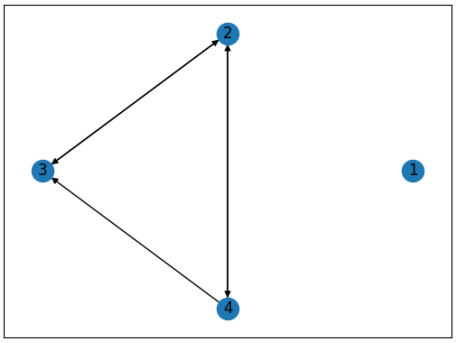
Zadanie 4.0.13 — 3 pkt

Napisz funkcję `dopelnienie(graf)`, przyjmującą jako jedyny argument `graf` reprezentowany jako słownik list, zwracającą *dopełnienie grafu*. *Dopełnienie grafu* to taki graf, który ma te same wierzchołki co oryginalny graf, ale ma krawędzie pomiędzy dowolnymi dwoma wierzchołkami wtedy i tylko wtedy, gdy oryginalny graf nie miał tam krawędzi.
Przykład:

```
> graf = {1: [2, 3, 4], 2: [1], 3: [1, 4], 4: [1]}
> dopelnienie(graf)
{1: [], 2: [3, 4], 3: [2], 4: [2, 3]}
```



(e) Graf oryginalny (`graf` z przykładu wyżej).

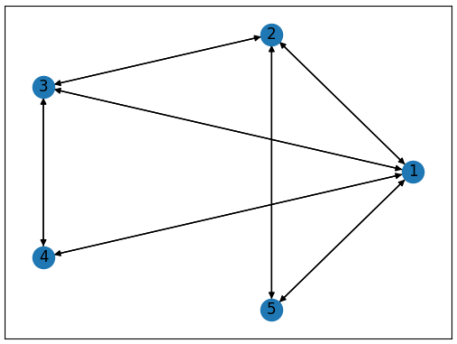


(f) Dopełnienie grafu z lewego rysunku.

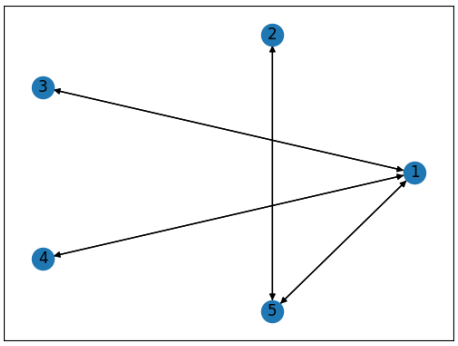
Zadanie 4.0.14 — 3 pkt

Napisz funkcję `drzewo_rozpinajace(graf)`, przyjmującą jako jedyny argument `graf` reprezentowany jako słownik list, zwracającą *drzewo rozpinające* tego grafu. *Drzewo rozpinające* to taki graf, który zawiera wszystkie wierzchołki grafu spójnego, ale tylko tyle krawędzi, aby był spójny.
Podpowiedź. W ogólnym przypadku istnieje wiele drzew rozpinających dla danego grafu. Wystarczy, że Twój program zwróci dowolne poprawne.
Przykład:

```
> graf = { 1: [2, 3, 4, 5], 2: [1, 3, 5], 3: [1, 2, 4], 4: [1, 3], 5: [1, 2] }
> drzewo_rozpinajace(graf)
{ 1: [3, 4, 5], 2: [5], 3: [1], 4: [1], 5: [1, 2] }
```



(g) Graf oryginalny (`graf` z przykładu wyżej).



(h) Przykładowe drzewo rozpinające grafu po lewej.

Podpowiedź: Kolejność przeglądania wierzchołków w algorytmie przeszukiwania wszerz lub w głąb wyznacza w naturalny sposób drzewo rozpinające.